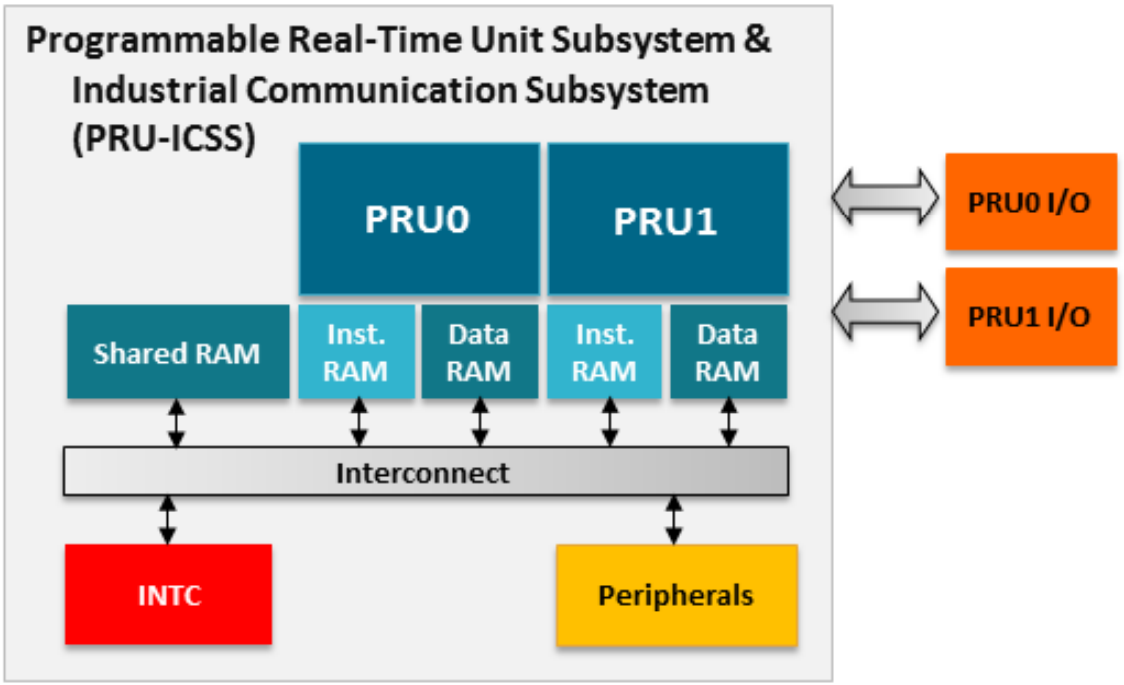
PRU Compiler Tips and Tricks

Embedded Processor Development Tools

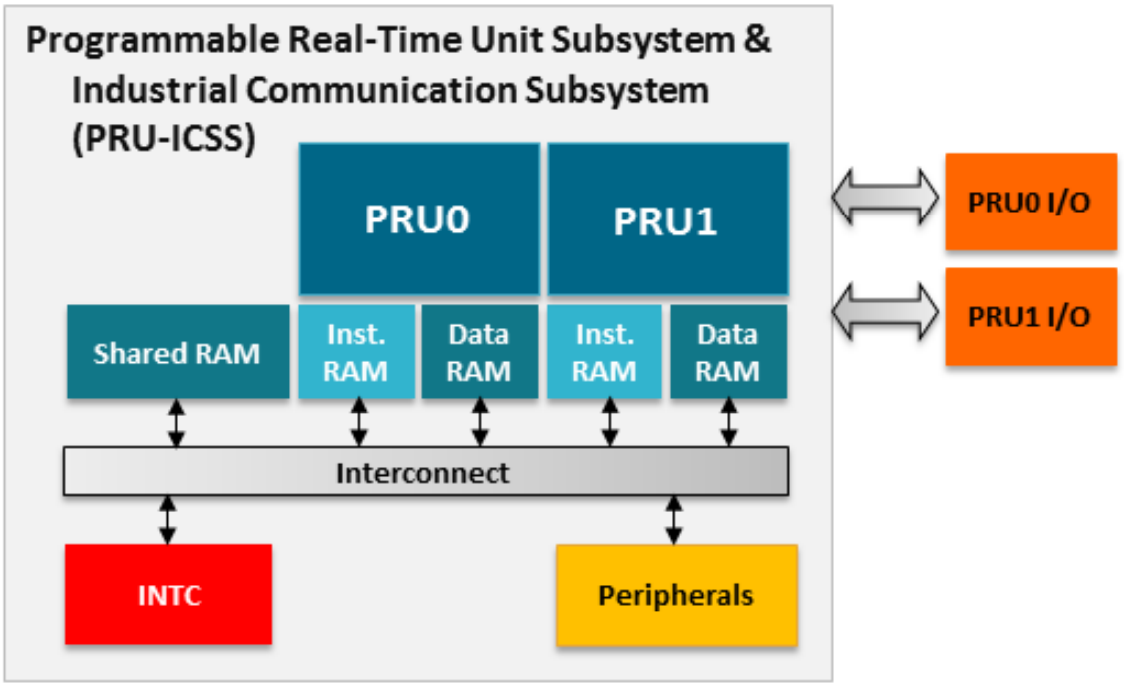
Where the PRU Compiler Fits In

- Industrial Communication Subsystem
- Most Sitara devices – such as AM335x on BeagleBone, AM437x, and AM572x – include the ICSS.



Where the PRU Compiler Fits In

- The ICSS contains two or four PRU cores.
- This presentation is about the C/C++ compiler for PRU.
- Compiler version 2.1.x



Why to Support Everything in C/C++

- Defining a language is hard. Even a subset.
- Users can easily handle the trade-off between language features and size, cycles, power, etc.
- Compiler testing is much better:
  - Huge commercial test suites can be used.
  - A subset language requires custom tests that are far less comprehensive.

EABI Only

- Only one Application Binary Interface (ABI) is supported.
- EABI: Extended ABI
- ELF object file format
- PRU executables can be read by Linux ELF tools.

```
$ readelf -S pru_exe.out
There are 27 section headers, starting at offset 0x3f54c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk
  [ 0]                   NULL            00000000 000000 000000 00      0
  [ 1] .text             PROGBITS        00000020 000034 0057e4 00  AX  0
  [ 2] .const            NOBITS          00000000 000000 000000 00      0
…
```

Memory Models

- Only one code memory model. All code must be within 16-bits.
- Two data memory models:
  - Default is near.
  - Near access more efficient than far.

```
near int n;
far  int f;
…
n = 10; // 3 instructions, 12 bytes
f = 10; // 4 instructions, 16 bytes
```

  - But near access is limited to addresses <= **`0xffff`**
  - far access has no limit

Data Types

- Scalar type sizes are typical:
  - char is 8-bits, short is 16-bits, int is 32-bits, etc
  - float is 32-bits, double is 64-bits
- Data pointers are 32-bits.
- Code pointers are 16-bits.
- Nothing is aligned. For example: 32-bit value may have an odd-byte address.

Signness of char

- Is plain **`char`** signed or unsigned? Default is unsigned.
- Two ways to override:
  - Write **`signed char`**
  - Use build option **`--plain_char=signed`**
- Recommendation: Avoid the whole thing.
  - **`#include <stdint.h>`**
  - Always use **`uint8_t`** or **`int8_t`**
  - Self documenting

Avoid Sign Extension

- Sign extension is required when 8-bit or 16-bit signed types are mixed with larger types.
- OK: Expressions that are all 8-bit signed or all 16-bit signed.
- Bad: Expressions which mix 8-bit signed or 16-bit signed with any larger type.
- Example:

```
s32 = s16;  // 7 instructions, including a branch
u32 = u16;  // 5 instructions
```

Avoid Floating Point

- No floating-point hardware.
- All floating-point operations are done in software
- Expensive: code size, cycles, power
- Use **`--float_operations_allowed=none`**
  - Get a diagnostic whenever float is used.

Avoid C++ Exceptions

- Disabled by default
- Enable with **`--exceptions`**
- EH: Exceptions handling
- Cannot mix EH code with non-EH code. Additional RTS library gets built.
- Exceptions are costly in size and cycles, even when never thrown.

Zero Overhead Loops

LOOP instruction used when:

- Build with **`--opt_level=2`** or higher.
- Loop bounds is 16-bit unsigned type, or constant <= **`0xffff`**
- Loop does not contain a function call or another loop.
- Other unusual conditions cannot be present.

```
uint16_t count16;
uint32 t count32;
for (i = 0; i < count16; ++i)     // zero overhead LOOP
for (i = 0; i < count32; ++i)     // normal overhead loop
for (i = 0; i <  0xffff; ++i)     // zero overhead LOOP
for (i = 0; i < 0x10000; ++i)     // normal overhead loop
```

Further Loop Improvement

- The following gets a zero overhead LOOP:

```
for (i = 0; i < count16; ++i)     // zero overhead LOOP
```

- But there is still a conditional branch before the loop which checks whether **`count16 == 0`**
- This branch can be avoided with **`MUST_ITERATE`**.

```
#pragma MUST_ITERATE(1)           // avoids check for count16 == 0
for (i = 0; i < count16; ++i)     // zero overhead LOOP
```

Intrinsics

- Intrinsics look and act like function calls.
- Implemented in one instruction:

```
void __halt();
void __delay_cycles(unsigned int cycles); // cycles must be constant
void __xin (unsigned int device_id, unsigned int base_register,
            unsigned int use_remapping, void &object);
// And 6 others
```

- Details in the compiler manual.
- See examples in the PRU Software Support Package.

PRU Software Support Package

- http://processors.wiki.ti.com/index.php/PRU-ICSS
- The Software Support Package is under the Getting Started Guide link:

```
/* PRU1_Direct_Connect.c */
void main() {
...
   /* Let PRU0 know that I am awake */
   __R31 = PRU1_PRU0_INTERRUPT+16;

   /* XFR registers R5-R10 from PRU0 to PRU1 */
   /* 14 is the device_id that signifies a PRU to PRU transfer */
   __xin(14, 5, 0, buf);

   /* Store register values back into DRAM */
   dmemBuf = buf;

   /* Halt the PRU core */
   __halt();
}
```

Linking

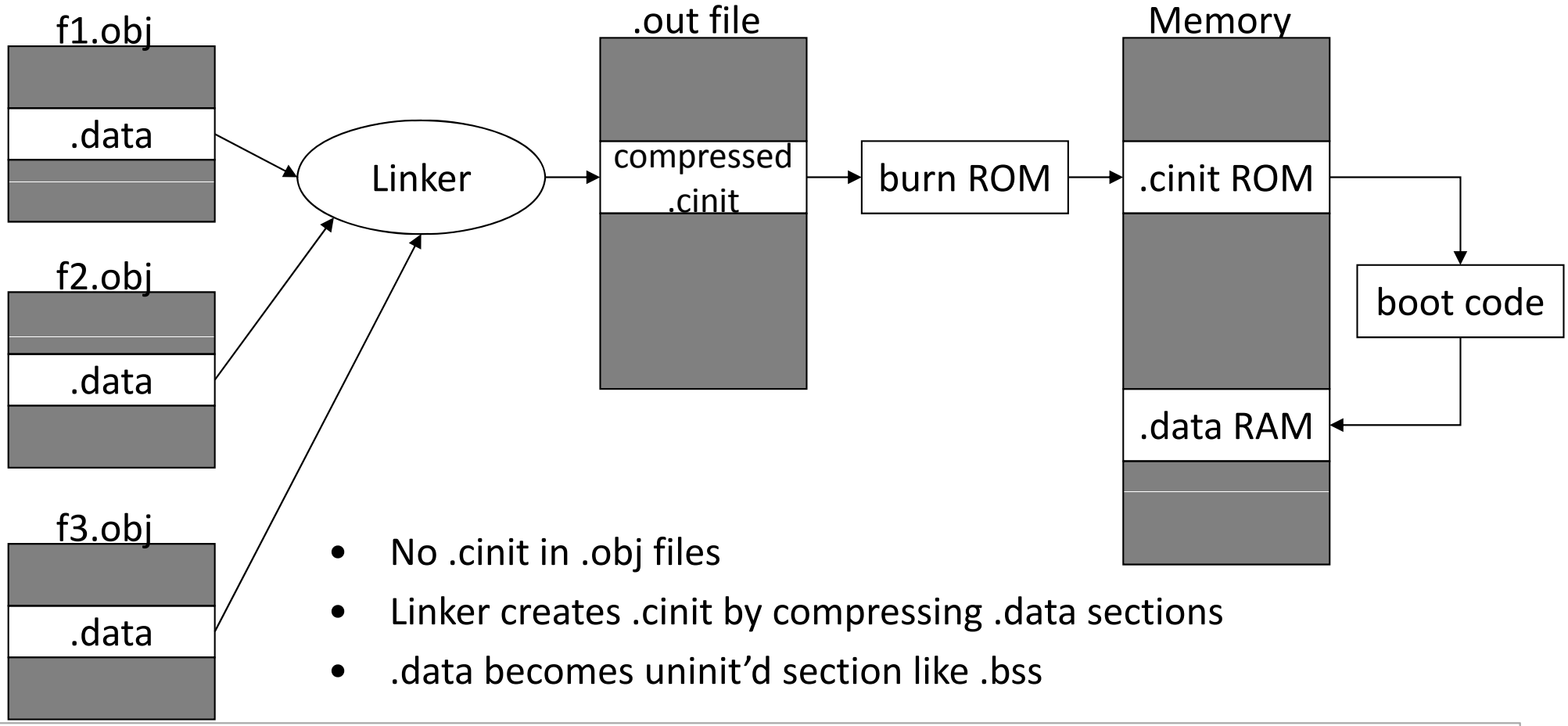
- How does this get initialized?

```
int global_var = 10;
```

- Where is the value 10 stored? How does it get copied to global_var before main starts to run?
- Two models: ROM Model and RAM Model

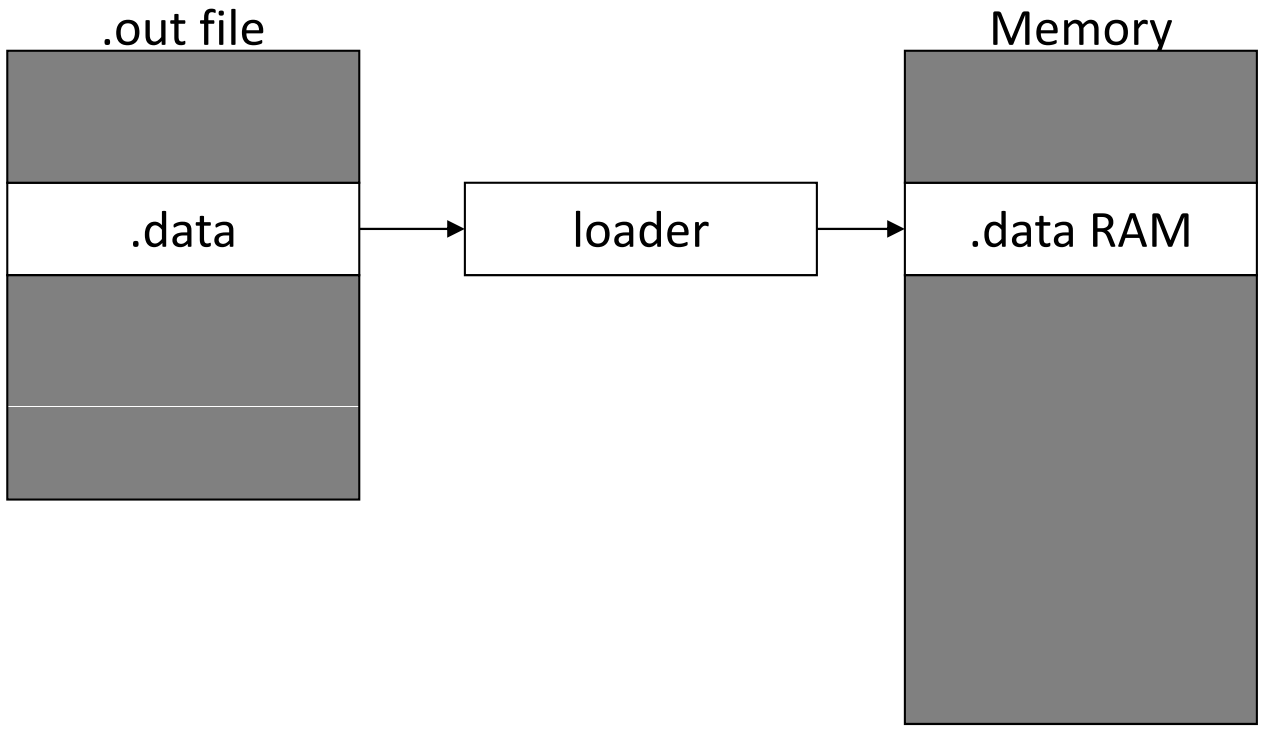
	ROM	RAM
Values stored in:	Target memory	.out file
Copied by:	Boot code	Loader

How --rom_model Works



- No .cinit in .obj files
- Linker creates .cinit by compressing .data sections
- .data becomes uninit'd section like .bss

How --ram_model Works



.data handled like any other initialized section.

Link with --ram_model

- ROM model uses about 2X the memory and more cycles at start-up.
- RAM model requires a loader in the system.
- But a loader is usually available for PRU systems:
  - Unlike most other embedded systems
  - Typically loaded by an ARM A8 running Linux
- So use **`--ram_model`**!
- Default remains **`--rom_model`**
  - Tools conservatively assume no loader is present

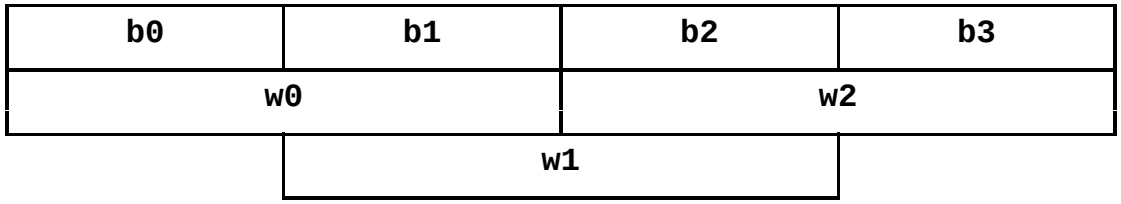
Memory Pages

- Not a single linear address space
- Memory is organized into two pages:
  - Code on page 0
  - Data on page 1
- Same address in each page has different contents.
- This is why tiobj2bin does not work for PRU:

```
/* AM335x_PRU.cmd */
MEMORY {
    PAGE 0:
        /* 8kB PRU0 Instruction RAM */
        PRU_IMEM      : org = 0x00000000 len = 0x00002000
    PAGE 1:
        /* 8kB PRU Data RAM 0_1 */
        PRU_DMEM_0_1  : org = 0x00000000 len = 0x00002000 CREGISTER=24
...
}
```

Register Fields

- Registers are 32-bits wide.
- Registers can be split into 16-bit fields or 8-bit fields.
- Fields are labeled:



- Examples: R3, R3.w2, R3.b1

Register Conventions

- R2: SP
- R3.w2: Return address
- R4: AP (rarely needed)
- R4-R13: Preserve if modified. All other registers are not preserved.
  - Compiler manual terms these *Save on Entry*
  - Others are termed *Save on Call*
- R14: Return value
- R14-R29: Argument registers
- R30-R31: Reserved to user

Save on Call Regs Are Not Saved

- R4-R13: Save on Entry (SOE)
- R0-R3, R14-R29: Save on Call (SOC)
- Expect to see SOC registers saved before a call. But you never see any saves. Why?
- Compiler never uses an SOC register for a value needed after the call.
- Values needed after the call are in SOE or on the stack.

Global Register Variables

- Special symbols **__R30** and **__R31** always correspond to control registers R30 and R31
- Declare volatile:

```
#include <stdint.h>      // for uint32_t
volatile register uint32_t __R30, __R31;
```

About the Volatile Keyword

- The volatile keyword appears more often in PRU code.
- A variable marked volatile:
  - May change due to something outside the scope of the single thread of execution the compiler knows.
  - Reads may cause some other change in the system.
- http://processors.wiki.ti.com/index.php/Volatile

How Arguments Are Passed

- Arguments are passed in registers R14-R29, then on the stack.
  - It is rare for arguments to get passed on the stack.
  - Thus, it is rare to need an AP.
- Scalars <= 32-bits
  - Packed into registers, using fields as needed
  - Always placed in single register
  - Later args may fill gaps left in earlier regs
- 64-bit scalars are passed in a register pair.
- Structures <= 64-bits are passed in registers.

Argument Passing Examples

```
fxn1(int a1, short a2, int a3, short a4);
//       R14    R15.w0     R16    R15.w2
```

- a4 does not use R17, but fills gap left in R15

```
fxn2(char a1, short a2, char a3);
//     R14.b0    R14.w1   R14.b3
```

- All args in R14, a2 in middle 16-bits

Constant Table Access

- From the PRU Software Support Package:

```
// pru_cfg.h
volatile __far pruCfg CT_CFG __attribute__((cregister("PRU_CFG", near), peripheral));
```

- Global structure **CT_CFG** maps to key configuration registers in constant table.
- Don’t write such code on your own.
- Barely counts as C code.
- The next few slides de-mystify some of it.
- Details are in the compiler manual.

Constant Table Access

```
// pru_cfg.h
volatile __far pruCfg CT_CFG __attribute__((cregister("PRU_CFG", near), peripheral));
```

- **`typedef`** is the name of the structure
- This **`typedef`** created in the same file:

```
// pru_cfg.h
typedef struct {
            // complicated set of unions, fields, etc
            // …
} pruCfg;
```

Constant Table Access

```
// pru_cfg.h
volatile __far pruCfg CT_CFG __attribute__((cregister("PRU_CFG", near), peripheral));
```

- Name of the variable being defined
- Typical usage in a C file:

```
            // PRU_access_const_table.c
CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
```

Constant Table Access

```
// pru_cfg.h
volatile __far pruCfg CT_CFG __attribute__((cregister("PRU_CFG", near), peripheral));
```

- Adds attributes to the variable.
- Extension to C borrowed from GCC
- Adds two attributes:
  - **`cregister`**
  - **`peripheral`**

Constant Table Access

```
// pru_cfg.h
volatile __far pruCfg CT_CFG __attribute__((cregister("PRU_CFG", near), peripheral));
```

Constant register attribute has two sub-attributes:

- **"PRU_CFG"** is the memory range name from linker command file.

```
/* AM335x_PRU.cmd */
MEMORY {
…
    PRU_CFG : org = 0x00026000 len = 0x00000044 CREGISTER=4
…
}
```

- **near** means access is near relative to the **cregister** peripheral.

Constant Table Access

```
// pru_cfg.h
volatile __far pruCfg CT_CFG __attribute__((cregister("PRU_CFG", near), peripheral));
```

CT_CFG

- Does not require allocation in memory like other global variables
- Can be included in multiple C files without causing redefinition errors

PRU-Specific Options

CCS sets these options when a new project is created:

- **`--silicon_version=[1-3]`**
  - Default value is 3.
  - Use the default value unless an expert tells you otherwise.
- **`--hardware_mac`**
  - This option enables the use of hardware multiply-accumulate.
  - Device data sheet indicates whether this feature is available.

Optimization

Option	Range of Optimization
--opt_level=off	None
--opt_level=0	Statements
--opt_level=1	Blocks
--opt_level=2	Functions
--opt_level=3	Files
--opt_level=4	Between files and libraries

- This is only a rough summary.
- Some level 0 and 1 optimizations range farther.

Initialized Sections

Name	Contents	N/F	RO
.text	Executable code	-	Y
.data	Initialized data	near	N
.fardata	Initialized data	far	N
.rodata	Read-only data	near	Y
.rofardata	Read-only data	far	Y
.cinit	Tables for init'ing global vars	-	Y
.init_array	Tables for C++ constructors	-	Y

- N/F : near or far access
- RO : May be in read-only memory
  - Not required

Uninitialized Sections

Name	Contents	N/F
.bss	Uninitialized data	near
.farbss	Uninitialized data	far
.stack	Stack	-
.sysmem	Heap for malloc, etc	-

Section names can be customized:

- Per function or variable
- Across a range of source lines
- #pragma and __attribute__ methods
- Details are in the compiler manual

For More Information

- Compiler e2e Forum

https://e2e.ti.com/support/development_tools/compiler/f/343

- Compiler Wiki

http://processors.wiki.ti.com/index.php?title=Category:Compiler

- PRU Software Support Package

http://processors.wiki.ti.com/index.php/PRU-ICSS